*Singing*SQL Presents:

The Case for Manual SQL Tuning

November, 2011

dantow@singingsql.com
www.singingsql.com

Overview

- Assumptions
- What is manual SQL tuning?
- A seemingly simple example
- Scenarios
- The role of manual tuning in solving any scenario

SQL Tuning Assumptions

- Oracle's Cost-based Optimizer (CBO) does a perfectly good job on most SQL, requiring no manual tuning for most SQL.
- The CBO *must* parse quickly, use the data and indexes that it has, make assumptions about what it does not know, and deliver *exactly* the result that the SQL calls for.
- On a small fraction of the SQL, the constraints on the CBO result in a performance problem.

What is Manual SQL Tuning?

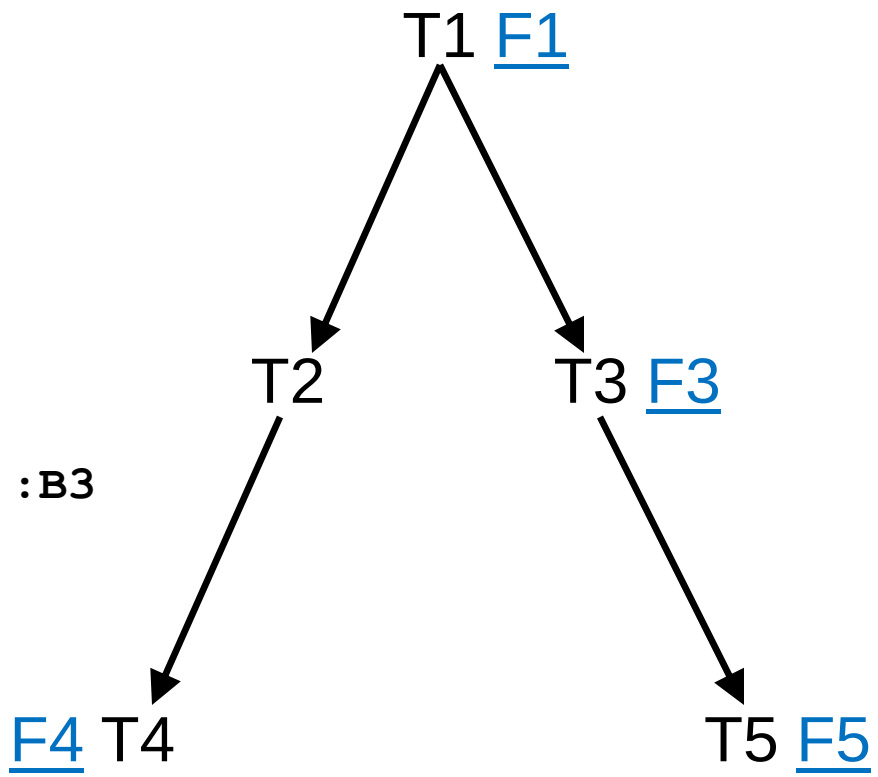
- *Find* SQL worth tuning, ignoring the great majority that already performs just fine.
- Find the true optimum execution plan (or at least one you verify is fast enough), manually, without the CBO's constraints or assumptions.
- Compare your manually chosen execution plan, and its resulting performance, with the CBO's plan and consider why the CBO did not select your plan, if it did not.
- Choose a solution that solves the problem.

A Seemingly Simple Example

```
SELECT ...
FROM
T1, T2, T3, T4, T5
WHERE T1.FKey2=T2.PKey2
AND T1.FKey3=T3.PKey3
AND T2.FKey4=T4.PKey4
AND T3.FKey5=T5.PKey5
AND T4.CCol4=:B1
AND T4.DCOL4 BETWEEN :B2 AND :B3
AND T1.CCol1=:B4
AND T3.CCol3=:B5
AND T5.CCol5=:B6
```

A Seemingly Simple Example

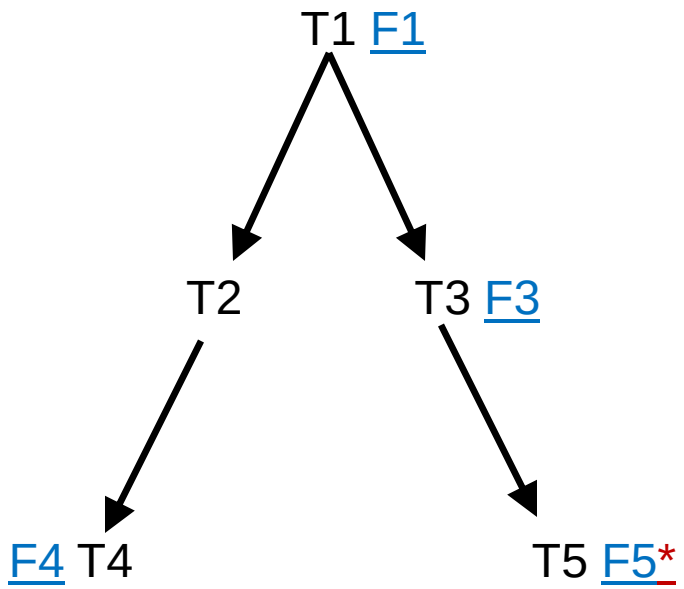
```
SELECT ...
FROM
T1, T2, T3, T4, T5
WHERE T1.FKey2=T2.PKey2
AND T1.FKey3=T3.PKey3
AND T2.FKey4=T4.PKey4
AND T3.FKey5=T5.PKey5
AND T4.CCol4=:B1
AND T4.DCOL4 BETWEEN :B2 AND :B3
AND T1.CCol1=:B4
AND T3.CCol3=:B5
AND T5.CCol5=:B6
```



All Scenarios: T1 is very large, much more poorly cached than the other tables.

Scenario #1

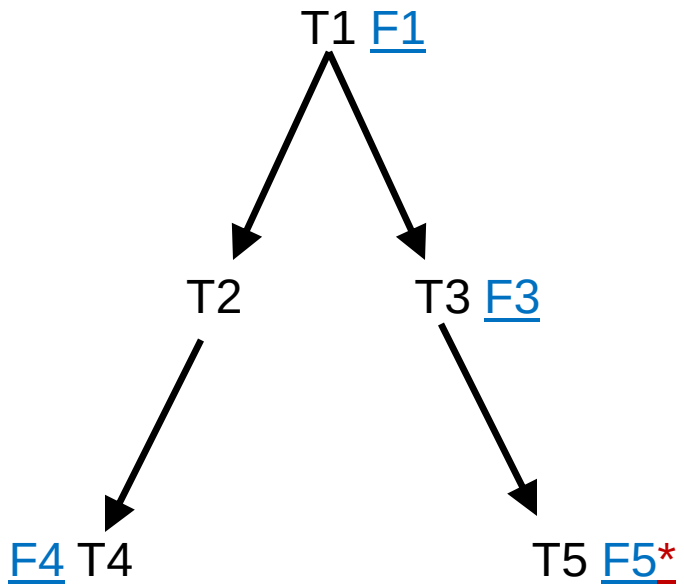
```
SELECT ...
FROM
T1, T2, T3, T4, T5
WHERE T1.FKey2=T2.PKey2
AND T1.FKey3=T3.PKey3
AND T2.FKey4=T4.PKey4
AND T3.FKey5=T5.PKey5
AND T4.CCol4=:B1
AND T4.DCOL4 BETWEEN :B2 AND :B3
AND T1.CCol1=:B4
AND T3.CCol3=:B5
AND T5.CCol5=:B6
```



T5.CCol5 has 2 values, but the value assigned to :B6 is super-rare, and the other filters are not nearly as selective, but the CBO plan now drives from the less selective filters on T4.

Scenario #1, Solution

```
SELECT ...
FROM
T1, T2, T3, T4, T5
WHERE T1.FKey2=T2.PKey2
AND T1.FKey3=T3.PKey3
AND T2.FKey4=T4.PKey4
AND T3.FKey5=T5.PKey5
AND T4.CCol4=:B1
AND T4.DCOL4 BETWEEN :B2 AND :B3
AND T1.CCol1=:B4
AND T3.CCol3=:B5
AND T5.CCol5='VeryRareValue'
```

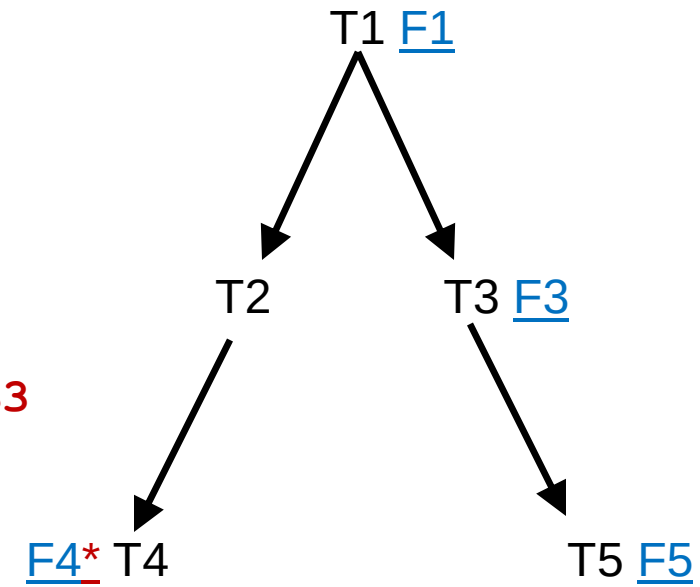


T5.CCol5 has 2 values, but the value assigned to :B6 is super-rare, and the other filters are not nearly as selective.

Solution1: :B6 should be hardcoded to the super-rare value, ideally, and we need a histogram on T5.CCol5. The CBO will do the right thing once provided with this added data.

Scenario #2

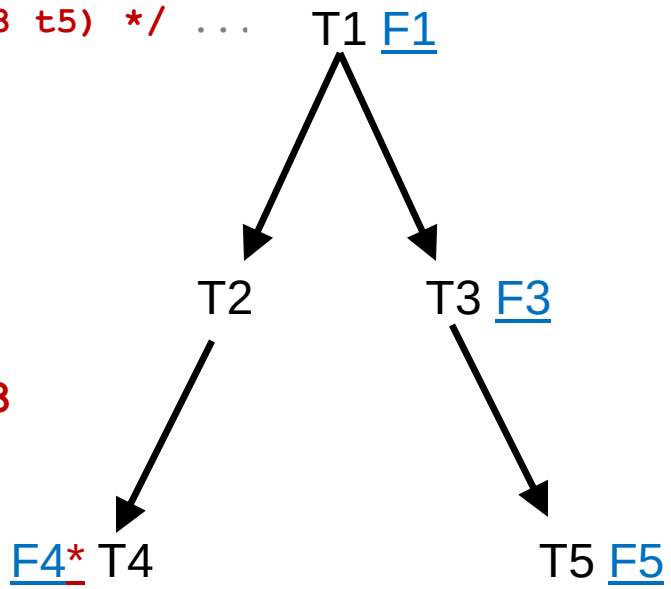
```
SELECT ...
FROM
T1, T2, T3, T4, T5
WHERE T1.FKey2=T2.PKey2
AND T1.FKey3=T3.PKey3
AND T2.FKey4=T4.PKey4
AND T3.FKey5=T5.PKey5
AND T4.CCol4=:B1
AND T4.DCOL4 BETWEEN :B2 AND :B3
AND T1.CCol1=:B4
AND T3.CCol3=:B5
AND T5.CCol5=:B6
```



Neither of the two filters on T4 are very selective by themselves, but they are highly anti-correlated, so they are super-rare, together, contrary to the CBO assumption of statistical independence between filters. As a result, the optimizer makes the wrong choice to drive from the moderately-selective filter on T5.

Scenario #2, Solution

```
SELECT /*+ leading(t4) use_nl(t4 t2 t1 t3 t5) */ ...
FROM
T1, T2, T3, T4, T5
WHERE T1.FKey2=T2.PKey2
AND T1.FKey3=T3.PKey3
AND T2.FKey4=T4.PKey4
AND T3.FKey5=T5.PKey5
AND T4.CCol4=:B1
AND T4.DCOL4 BETWEEN :B2 AND :B3
AND T1.CCol1=:B4
AND T3.CCol3=:B5
AND T5.CCol5=:B6
```

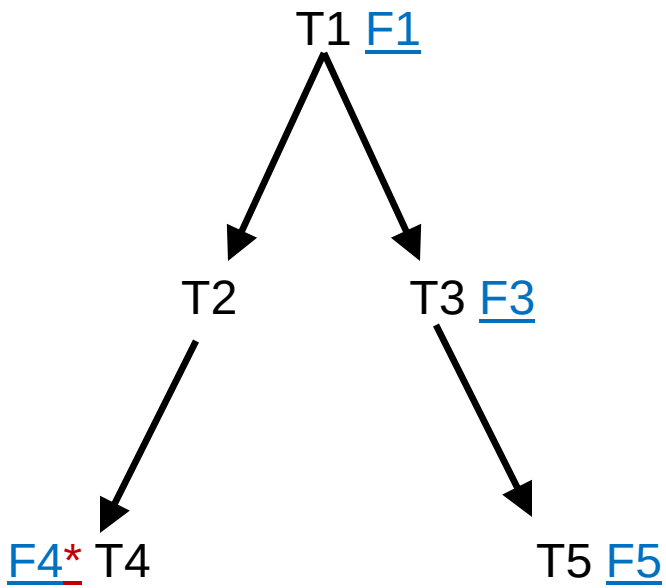


The filters on T4 are highly anti-correlated, so they are super-rare, together.

Solution2_1: Use hints or a stored outline to force leading access to T4 and nested-loops to the rest.

Scenario #2, Solution

```
SELECT ...
FROM
T1, T2, T3, T4, T5
WHERE T1.FKey2=T2.PKey2
AND T1.FKey3=T3.PKey3
AND T2.FKey4=T4.PKey4
AND T3.FKey5=T5.PKey5
AND T4.CCol4=:B1
AND T4.DCOL4 BETWEEN :B2 AND :B3
AND T1.CCol1=:B4
AND T3.CCol3=:B5
AND T5.CCol5=:B6
```

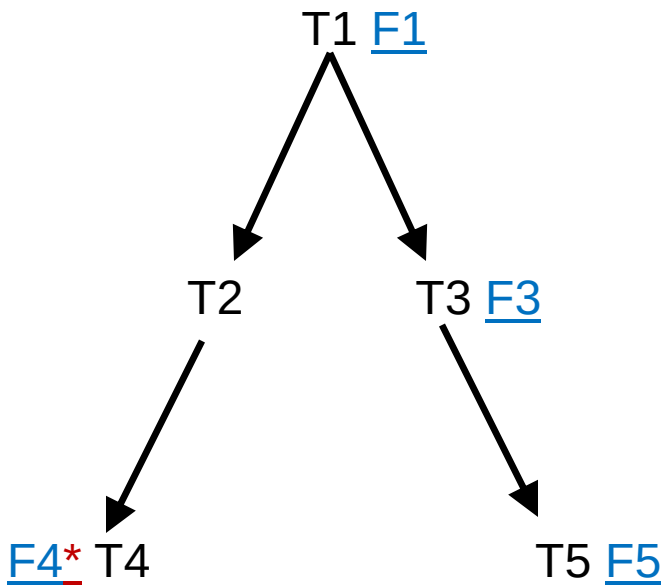


The filters on T4 are highly anti-correlated, so they are super-rare, together.

Solution2_2: Use dynamic sampling at a higher-than-default level so Oracle picks up the anti-correlated conditions at parse time. (This adds cost for every hard parse of this SQL, though.)

Scenario #2, Solution

```
SELECT ...
FROM
T1, T2, T3, T4, T5
WHERE T1.FKey2=T2.PKey2
AND T1.FKey3=T3.PKey3
AND T2.FKey4=T4.PKey4
AND T3.FKey5=T5.PKey5
--AND T4.CCol4=:B1
AND T4.DCOLX BETWEEN :B2 AND :B3
AND T1.CCol1=:B4
AND T3.CCol3=:B5
AND T5.CCol5=:B6
```

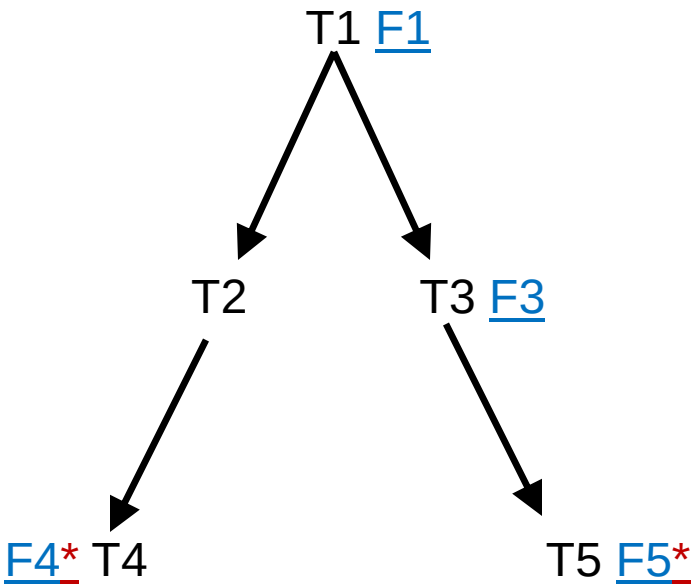


The filters on T4 are highly anti-correlated, so they are super-rare, together.

Solution2_3: Denormalize T4 with a new column DColX=DECODE(CCol4,'<ValueGiven:B1>',DCol4,NULL), and replace the T4 filters with DColX BETWEEN :B2 AND :B3. Consider a histogram and index on DColX. Preferably use triggers to populate DColX. (In an advanced-enough version, a functional index could replace DColX.)

Scenario #3

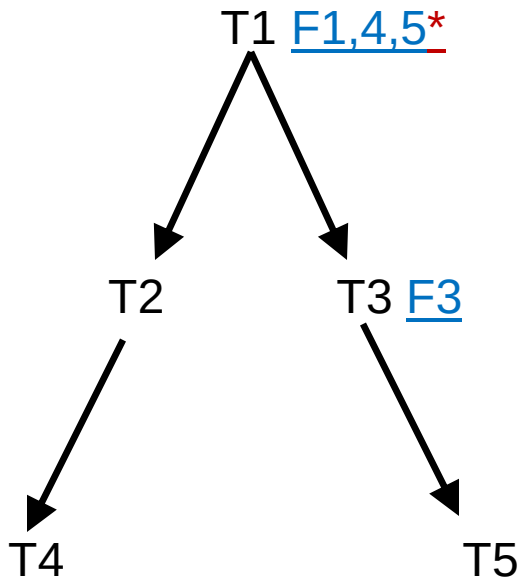
```
SELECT ...
FROM
T1, T2, T3, T4, T5
WHERE T1.FKey2=T2.PKey2
AND T1.FKey3=T3.PKey3
AND T2.FKey4=T4.PKey4
AND T3.FKey5=T5.PKey5
AND T4.CCol4=:B1
AND T4.DCOL4 BETWEEN :B2 AND :B3
AND T1.CCol1=:B4
AND T3.CCol3=:B5
AND T5.CCol5=:B6
```



The only moderately selective conditions are the conditions on CCol4 and CCol5 on T4 and T5, but neither one of these is selective enough, by itself, although, together as independently selective conditions, they reduce the result to a small number of rows. The SQL is very high-load, justifying denormalization if necessary to solve the problem.

Scenario #3, Solution

```
SELECT ...
FROM
T1, T2, T3, T4, T5
WHERE T1.FKey2=T2.PKey2
AND T1.FKey3=T3.PKey3
AND T2.FKey4=T4.PKey4
AND T3.FKey5=T5.PKey5
AND T1.CCol4=:B1
AND T4.DCOL4 BETWEEN :B2 AND :B3
AND T1.CCol1=:B4
AND T3.CCol3=:B5
AND T1.CCol5=:B6
```

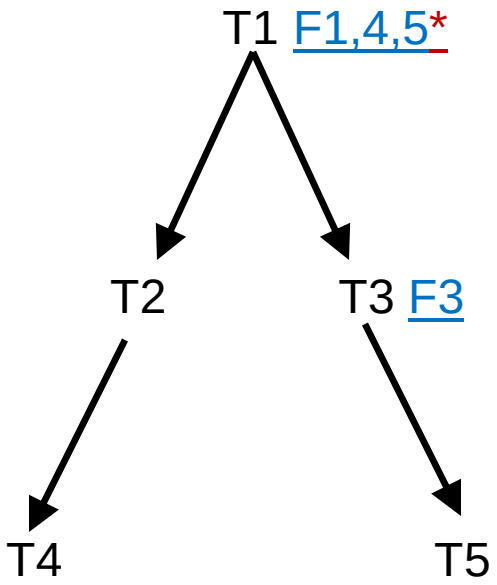


The only moderately selective conditions are the conditions on CCol4 and CCol5 on T4 and T5, but neither one of these is selective enough, by itself.

Solution3_1: Denormalize CCol4 and CCol5 to T1, using triggers to keep the new columns rigorously in sync, and create an index on T1(CCol4,CCol5) and alter the conditions on these columns to reference the new columns T1.CCol4 and T1.CCol5. The CBO will then do the right thing.

Scenario #3, Solution

```
SELECT ...
FROM
T1, T2, T3, T4, T5
WHERE T1.FKey2=T2.PKey2
AND T1.FKey3=T3.PKey3
AND T2.FKey4=T4.PKey4
AND T3.FKey5=T5.PKey5
AND T1.CCol4=:B1
AND T4.DCOL4 BETWEEN :B2 AND :B3
AND T1.CCol1=:B4
AND T3.CCol3=:B5
AND T1.CCol5=:B6
```



The only moderately selective conditions are the conditions on CCol4 and CCol5 on T4 and T5, but neither one of these is selective enough, by itself.

Solution3_2: Discover that the denormalization needed for Solution3_1 already exists, and use it.

Scenario #3, Solution

```
SELECT ...
FROM
T1, T2, T3, T4, T5
WHERE T1.FKey2=T2.PKey2
AND T1.FKey3=T3.PKey3
AND T2.FKey4=T4.PKey4
AND T3.FKey5=T5.PKey5
AND T4.CCol4=:B1
AND T4.DCOL4 BETWEEN :B2 AND :B3
AND T1.CCol1=:B4
AND T3.CCol3=:B5
AND T5.CCol5=:B6
```

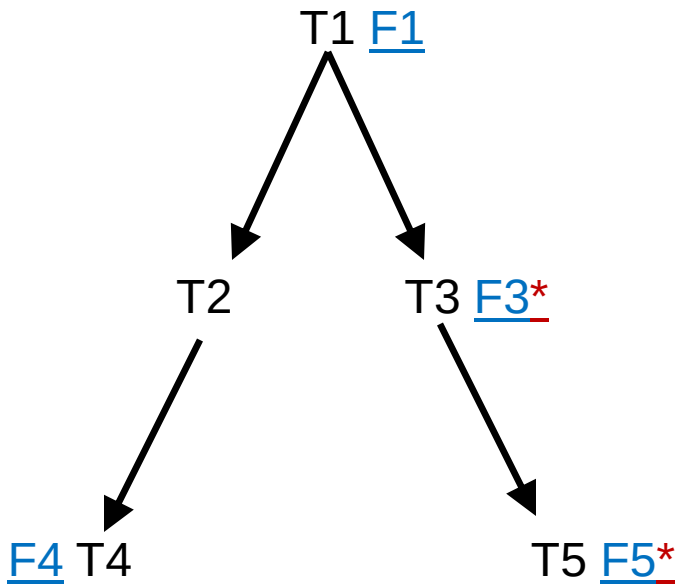
NewMatView F

The only moderately selective conditions are the conditions on CCol4 and CCol5 on T4 and T5, but neither one of these is selective enough, by itself.

Solution3_3: Assuming that not-quite-perfectly-up-to-date data is OK, consider a materialized view of this 5-way join, including all the columns this SQL needs, and allow query-rewrite, here. Index (CCol4, CCol5) on this materialized view, and the CBO will do the right thing with the existing SQL.

Scenario #4

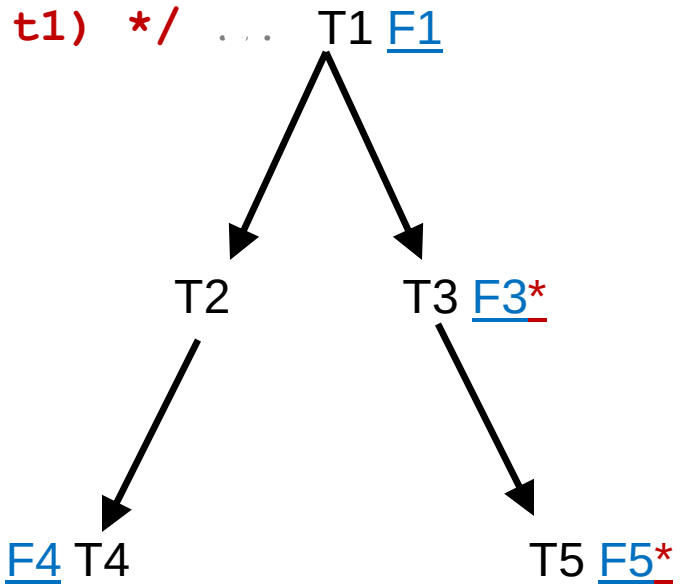
```
SELECT ...
FROM
T1, T2, T3, T4, T5
WHERE T1.FKey2=T2.PKey2
AND T1.FKey3=T3.PKey3
AND T2.FKey4=T4.PKey4
AND T3.FKey5=T5.PKey5
AND T4.CCol4=:B1
AND T4.DCOL4 BETWEEN :B2 AND :B3
AND T1.CCol1=:B4
AND T3.CCol3=:B5
AND T5.CCol5=:B6
```



The conditions on CCol3 and CCol5 are highly anticorrelated, so that the combination of these conditions is highly selective even though the combination (assuming statistical independence of filters, as the optimizer does assume) looks unselective. The filter on CCol4 looks more selective, by itself, so the optimizer chooses to reach T1 through the wrong query branch, at much higher runtime.

Scenario #4, Solution

```
SELECT /*+ leading(t3 t5) use_nl(t3 t5 t1) */ ...
FROM
T1, T2, T3, T4, T5
WHERE T1.FKey2=T2.PKey2
AND T1.FKey3=T3.PKey3
AND T2.FKey4=T4.PKey4
AND T3.FKey5=T5.PKey5
AND T4.CCol4=:B1
AND T4.DCOL4 BETWEEN :B2 AND :B3
AND T1.CCol1=:B4
AND T3.CCol3=:B5
AND T5.CCol5=:B6
```

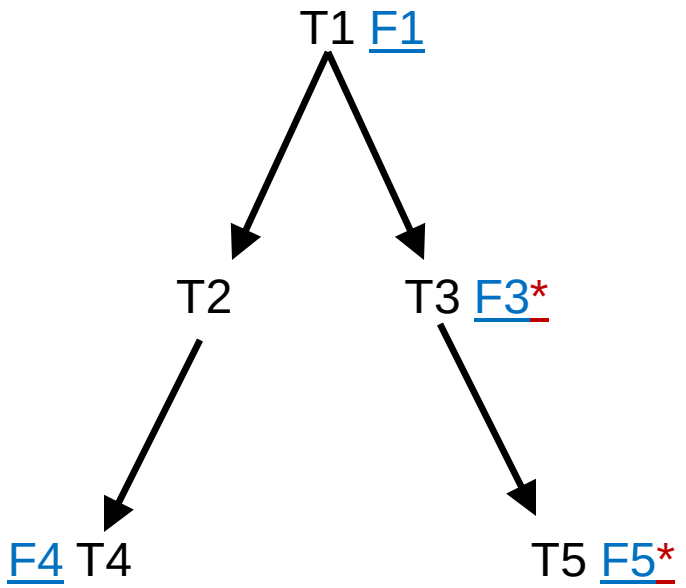


The conditions on CCol3 and CCol5 are highly anticorrelated.

Solution4: Force a join order, with hints or with a stored outline, that begins with a join of T3 and T5, and follows the join keys, using nested loops, from there. Dynamic sampling cannot find this anticorrelation. No change at the database level apart from artificial, *incorrect* stats can bring the optimizer to this choice without forcing it.

Scenario #5

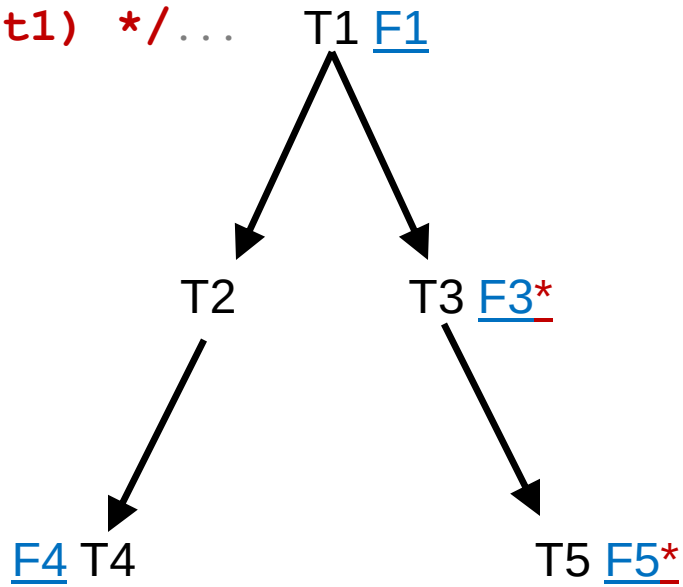
```
SELECT ...
FROM
T1, T2, T3, T4, T5
WHERE T1.FKey2=T2.PKey2
AND T1.FKey3=T3.PKey3
AND T2.FKey4=T4.PKey4
AND T3.FKey5=T5.PKey5
AND T4.CCol4=:B1
AND T4.DCOL4 BETWEEN :B2 AND :B3
AND T1.CCol1=:B4
AND T3.CCol3=:B5
AND T5.CCol5=:B6
```



After driving (correctly) to T3 as the leading table, the optimizer chooses to join to T1 before joining to T5 because a little less logical I/O is required that way, and the optimizer cost function assumes (roughly) that all logical I/O is equally likely to result in physical I/O, In fact, however, T5 is far smaller and better cached than T1, so the join to T5 first, picking up its useful filter before joining to T1, results in much less physical I/O and therefore a much faster runtime.

Scenario #5, Solution

```
SELECT /*+ leading(t3 t5) use_nl(t3 t5 t1) */...
FROM
T1, T2, T3, T4, T5
WHERE T1.FKey2=T2.PKey2
AND T1.FKey3=T3.PKey3
AND T2.FKey4=T4.PKey4
AND T3.FKey5=T5.PKey5
AND T4.CCol4=:B1
AND T4.DCOL4 BETWEEN :B2 AND :B3
AND T1.CCol1=:B4
AND T3.CCol3=:B5
AND T5.CCol5=:B6
```

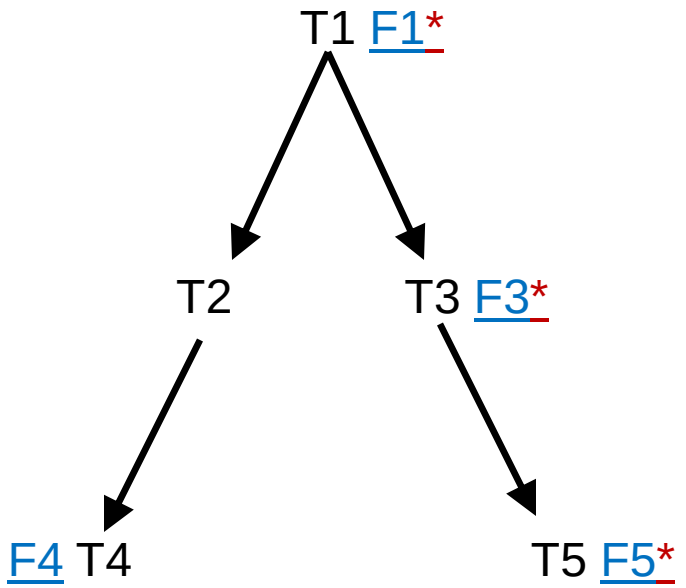


We need T3, T5, T1, but we are getting T3, T1, T5 because the cost function misestimates relative hit ratios.

Solution5: Force a join order, with hints or with a stored outline, and force nested loops to join keys, from there. No change at the database level apart from artificial, *incorrect* stats can solve this without overriding the natural optimizer choice, and incorrect stats would handicap the optimizer when optimizing other SQL, likely causing more harm than benefit.

Scenario #6

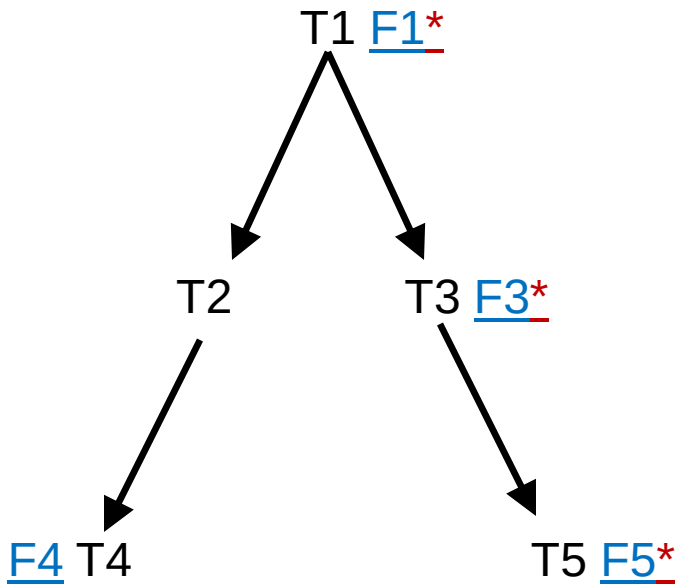
```
SELECT ...
FROM
T1, T2, T3, T4, T5
WHERE T1.FKey2=T2.PKey2
AND T1.FKey3=T3.PKey3
AND T2.FKey4=T4.PKey4
AND T3.FKey5=T5.PKey5
AND T4.CCol4=:B1
AND T4.DCOL4 BETWEEN :B2 AND :B3
AND T1.CCol1=:B4
AND T3.CCol3=:B5
AND T5.CCol5=:B6
```



The only way to reach T1 with low-enough physical I/O is reach it through the branch from T3 and to pick up the moderately selective T1 filter on CCol1 at the same time that it follows nested loops from FKey3, but it has no index on FKey3 at all. :B4 is always set to the same value, <B4Val>.

Scenario #6, Solution

```
SELECT ...
FROM
T1, T2, T3, T4, T5
WHERE T1.FKey2=T2.PKey2
AND T1.FKey3=T3.PKey3
AND T2.FKey4=T4.PKey4
AND T3.FKey5=T5.PKey5
AND T4.CCol4=:B1
AND T4.DCOL4 BETWEEN :B2 AND :B3
AND T1.CCol1=:B4
AND T3.CCol3=:B5
AND T5.CCol5=:B6
```

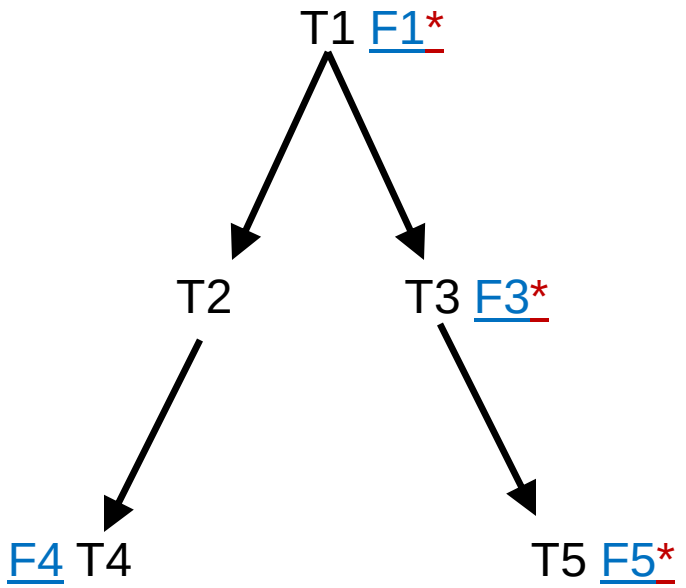


We need to reach T1 through the not-yet-indexed foreign key and F1 filter at the same time.

Solution6_1: Create a new index on T1(FKey3,CCol1). This index may be useful for other queries that will reach T1 through Fkey3, alone, and it will also achieve the objective of picking up the filter before reaching T1 in this particular query, but at the cost of a very large new two-column index. If CCol1 is changed routinely during the life of T1 rows (as for a status column, for example), maintenance of this index will be expensive.

Scenario #6, Solution

```
SELECT ...
FROM
T1, T2, T3, T4, T5
WHERE T1.FKey2=T2.PKey2
AND T1.FKey3=T3.PKey3
AND T2.FKey4=T4.PKey4
AND T3.FKey5=T5.PKey5
AND T4.CCol4=:B1
AND T4.DCOL4 BETWEEN :B2 AND :B3
AND T1.CCol1=:B4
AND T3.CCol3=:B5
AND T5.CCol5=:B6
```

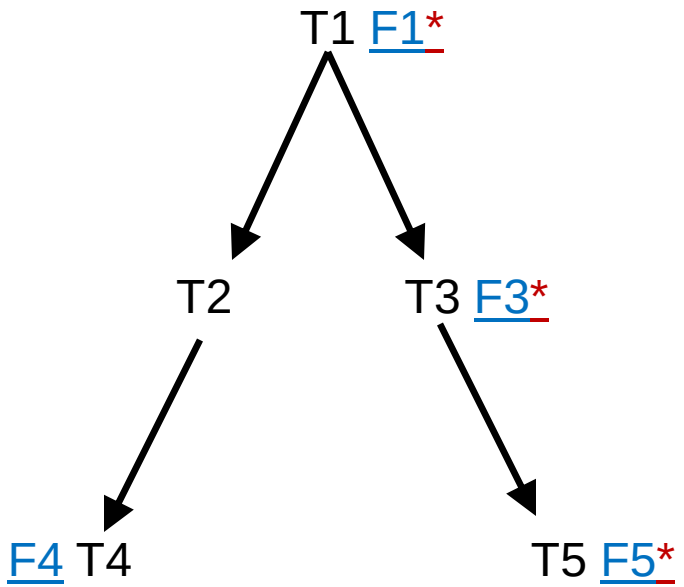


We need to reach T1 through the not-yet-indexed foreign key and F1 filter at the same time.

Solution6_2: Create a new index on T1(CCol1,FKey3). The index entries needed for this query will be stored closer together, resulting in better self-caching during query execution and less physical I/O to this index. If CCol1 is changed routinely during the life of T1 rows, maintenance of this index will be very expensive, with changes to CCol1 resulting in putting the new index entry far from the old entry, badly fragmenting the index over time.

Scenario #6, Solution

```
SELECT ...
FROM
T1, T2, T3, T4, T5
WHERE T1.FKey2=T2.PKey2
--AND T1.FKey3=T3.PKey3
AND T2.FKey4=T4.PKey4
AND T3.FKey5=T5.PKey5
AND T4.CCol4=:B1
AND T4.DCOL4 BETWEEN :B2 AND :B3
--AND T1.CCol1=:B4
AND T3.CCol3=:B5
AND T5.CCol5=:B6
AND DECODE(T1.CCol1,'<B4Val>',T1.FKey3,NULL)=T3.PKey3
```



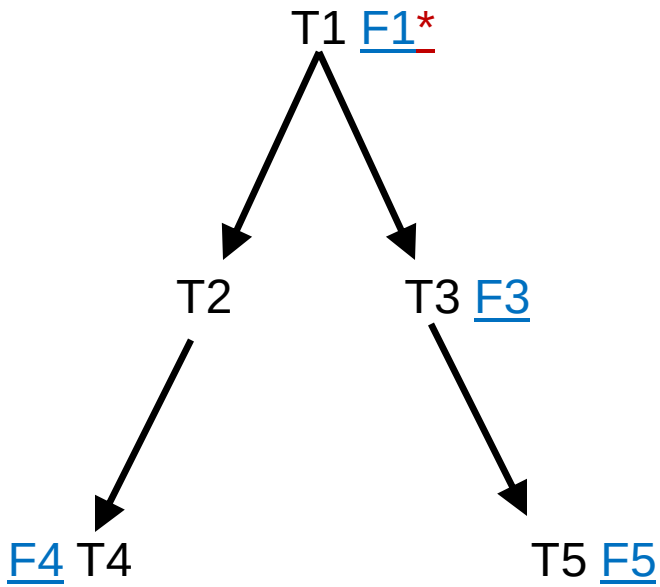
We need to reach T1 through the not-yet-indexed foreign key and F1 filter at the same time.

Solution6_3: Create a new index on T1(DECODE(CCol1,'<B4Val>',FKey3,NULL)) and replace the T1 join and filter with AND DECODE(T1.CCol1,'<B4Val>',T1.FKey3,NULL)=T3.PKey3

This index is much more compact and easy to maintain.

Scenario #7

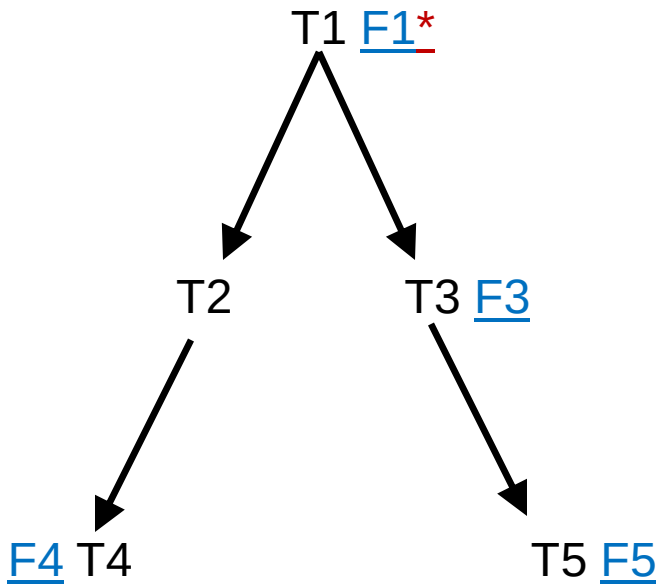
```
SELECT ...
FROM
T1, T2, T3, T4, T5
WHERE T1.FKey2=T2.PKey2
AND T1.FKey3=T3.PKey3
AND T2.FKey4=T4.PKey4
AND T3.FKey5=T5.PKey5
AND T4.CCol4=:B1
AND T4.DCOL4 BETWEEN :B2 AND :B3
AND T1.CCol1=:B4
AND T3.CCol3=:B5
AND T5.CCol5=:B6
```



The condition on CCol1 is the correct driving condition, but the optimizer cannot use the index on CCol1 because :B4 is number-type, while CCol1 is a varchar2, so there is an implicit type conversion on CCol1 that disables use of the existing index on T1.CCol1.

Scenario #7, Solution

```
SELECT ...
FROM
T1, T2, T3, T4, T5
WHERE T1.FKey2=T2.PKey2
AND T1.FKey3=T3.PKey3
AND T2.FKey4=T4.PKey4
AND T3.FKey5=T5.PKey5
AND T4.CCol4=:B1
AND T4.DCOL4 BETWEEN :B2 AND :B3
AND T1.CCol1=:B4
AND T3.CCol3=:B5
AND T5.CCol5=:B6
```

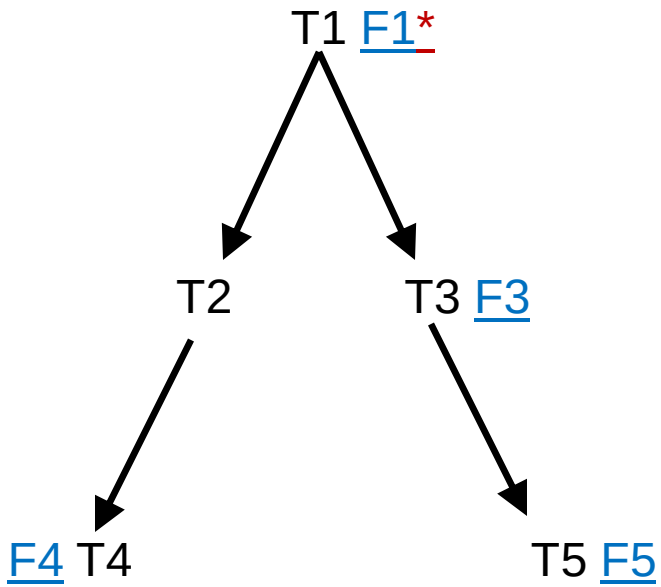


The condition on CCol1 has an implicit type conversion disabling an index.

Solution7_1: Change :B4 to character-type, but watch out for a subtle change in functionality and consider a new column constraint on CCol1.

Scenario #7, Solution

```
SELECT ...
FROM
T1, T2, T3, T4, T5
WHERE T1.FKey2=T2.PKey2
AND T1.FKey3=T3.PKey3
AND T2.FKey4=T4.PKey4
AND T3.FKey5=T5.PKey5
AND T4.CCol4=:B1
AND T4.DCOL4 BETWEEN :B2 AND :B3
AND T1.CCol1=:B4
AND T3.CCol3=:B5
AND T5.CCol5=:B6
```

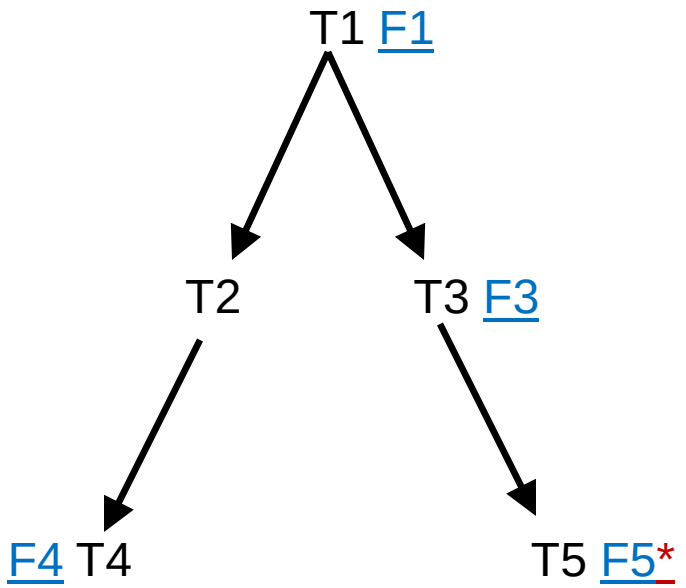


The condition on CCol1 has an implicit type conversion disabling an index.

Solution7_2: Create a new functional index on T1(to_number(CCol1)). This fixes the problem without change to the SQL, but it is wasteful since we already have an index on T1(CCol1), and Solution7_1 requires no new index.

Scenario #8

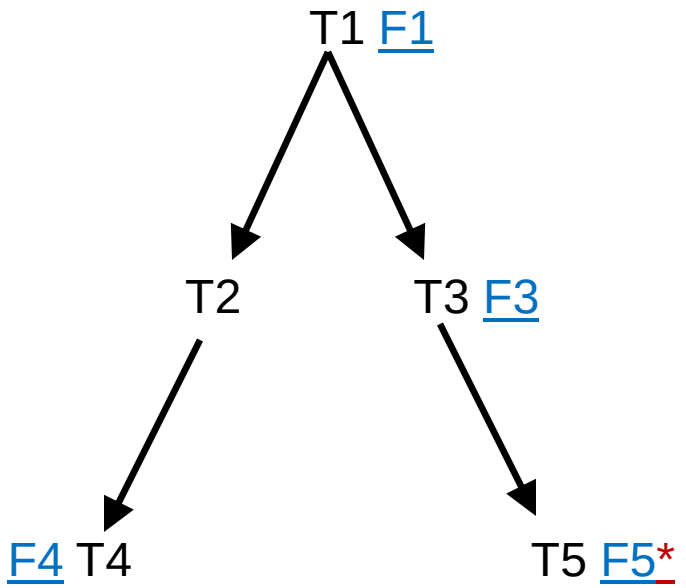
```
SELECT ...
FROM
T1, T2, T3, T4, T5
WHERE T1.FKey2=T2.PKey2
AND T1.FKey3=T3.PKey3
AND T2.FKey4=T4.PKey4
AND T3.FKey5=T5.PKey5
AND T4.CCol4=:B1
AND T4.DCOL4 BETWEEN :B2 AND :B3
AND T1.CCol1=:B4
AND T3.CCol3=:B5
AND T5.CCol5=:B6
```



T1.FKey3 is a varchar2, but T3.PKey3 is number-type and the most selective filter in the query, by far, is the condition on the indexed column T5.CCol5. There is already an index on T1(FKey3).

Scenario #8, Solution

```
SELECT ...
FROM
T1, T2, T3, T4, T5
WHERE T1.FKey2=T2.PKey2
AND T1.FKey3=TO_CHAR(T3.PKey3)
AND T2.FKey4=T4.PKey4
AND T3.FKey5=T5.PKey5
AND T4.CCol4=:B1
AND T4.DCOL4 BETWEEN :B2 AND :B3
AND T1.CCol1=:B4
AND T3.CCol3=:B5
AND T5.CCol5=:B6
```

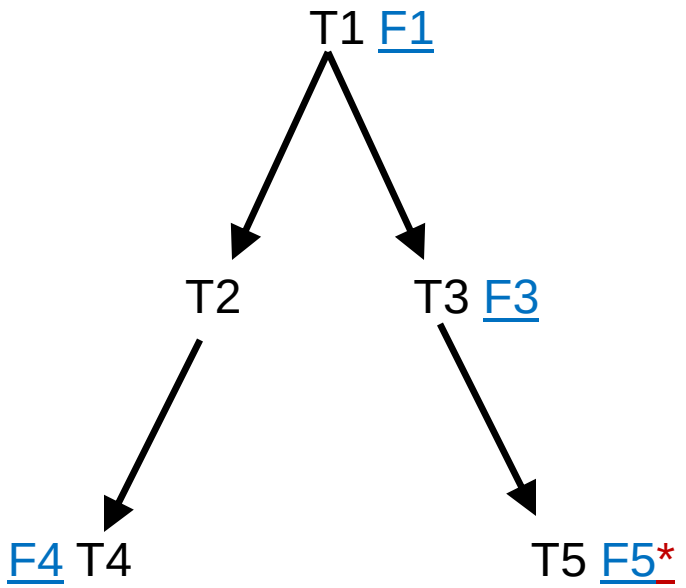


The join of T1 and T3 has an implicit type conversion that disables use of the foreign-key index.

Solution8_1: Make the type conversion explicit on the other side, but note that this subtly changes functionality and may call for a new column constraint on FKey3.

Scenario #8, Solution

```
SELECT ...
FROM
T1, T2, T3, T4, T5
WHERE T1.FKey2=T2.PKey2
AND T1.FKey3=T3.PKey3
AND T2.FKey4=T4.PKey4
AND T3.FKey5=T5.PKey5
AND T4.CCol4=:B1
AND T4.DCOL4 BETWEEN :B2 AND :B3
AND T1.CCol1=:B4
AND T3.CCol3=:B5
AND T5.CCol5=:B6
```

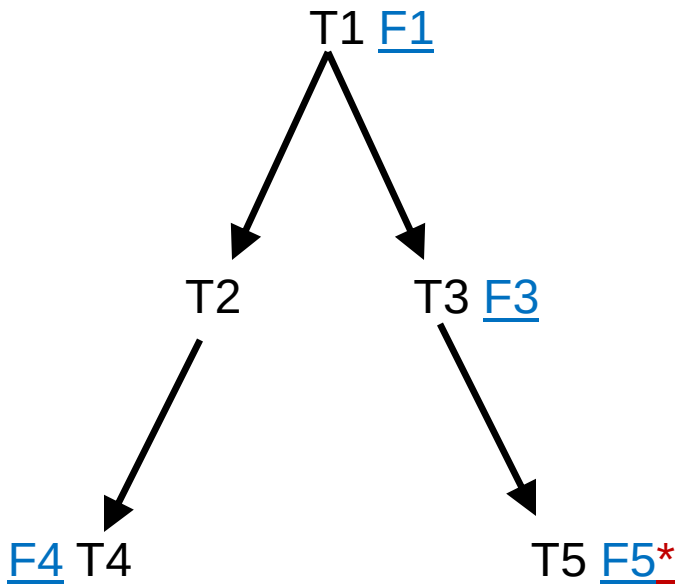


The join of T1 and T3 has an implicit type conversion that disables use of the foreign-key index.

Solution8_2: Create a new index on T1(TO_NUMBER(FKey3)), but note that this has functional implications and needs a large new index that Solution8_1 doesn't need.

Scenario #8, Solution

```
SELECT ...
FROM
T1, T2, T3, T4, T5
WHERE T1.FKey2=T2.PKey2
AND T1.FKey3=T3.PKey3
AND T2.FKey4=T4.PKey4
AND T3.FKey5=T5.PKey5
AND T4.CCol4=:B1
AND T4.DCOL4 BETWEEN :B2 AND :B3
AND T1.CCol1=:B4
AND T3.CCol3=:B5
AND T5.CCol5=:B6
```

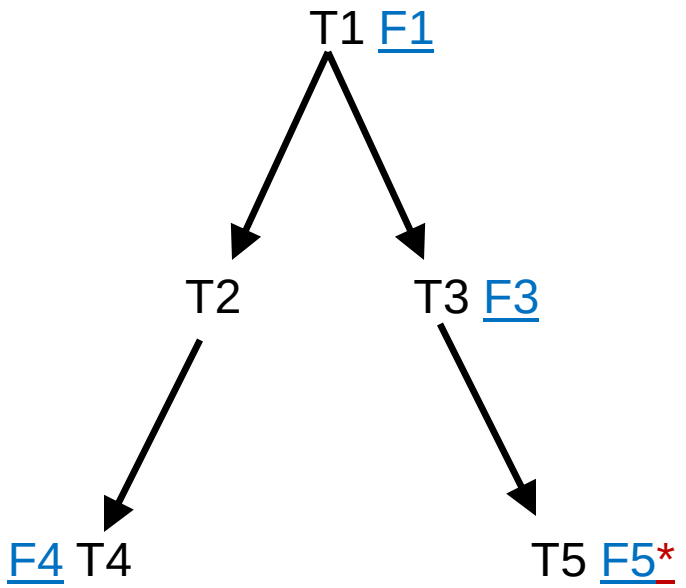


The join of T1 and T3 has an implicit type conversion that disables use of the foreign-key index.

Solution8_3: Change the database design, if possible, to make the keys type-consistent and migrate to the new design.

Scenario #9

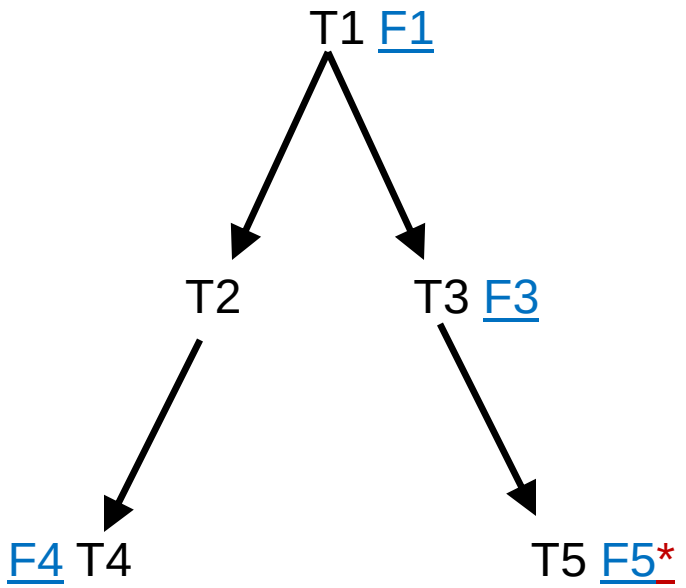
```
SELECT ...
FROM
T1, T2, T3, T4, T5
WHERE T1.FKey2=T2.PKey2
AND T1.FKey3=T3.PKey3
AND T2.FKey4=T4.PKey4
AND T3.FKey5=T5.PKey5
AND T4.CCol4=:B1
AND T4.DCOL4 BETWEEN :B2 AND :B3
AND T1.CCol1=:B4
AND T3.CCol3=:B5
AND T5.CCol5=:B6
```



The optimizer already chooses an optimally efficient execution plan. We are getting a very high rowcount with a minimum number of logical and physical I/Os per row, already. Performance is still unacceptable, though, because runtime is very high.

Scenario #9, Solution

```
SELECT ...
FROM
T1, T2, T3, T4, T5
WHERE T1.FKey2=T2.PKey2
AND T1.FKey3=T3.PKey3
AND T2.FKey4=T4.PKey4
AND T3.FKey5=T5.PKey5
AND T4.CCol4=:B1
AND T4.DCOL4 BETWEEN :B2 AND :B3
AND T1.CCol1=:B4
AND T3.CCol3=:B5
AND T5.CCol5=:B6
```

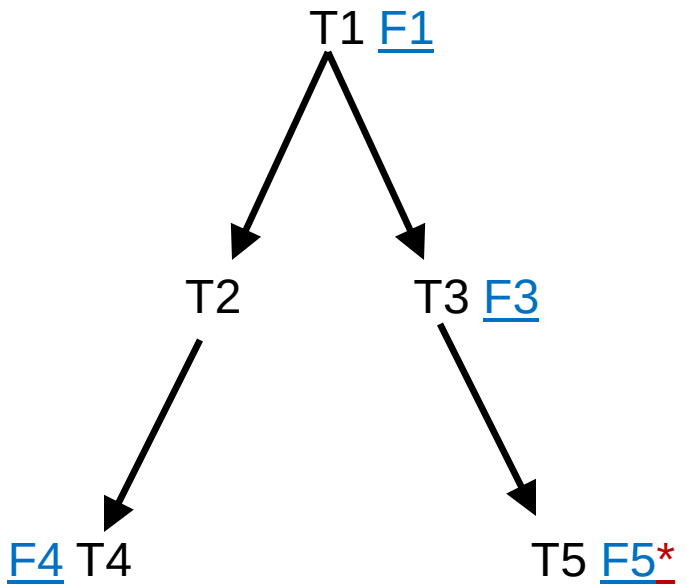


The plan is already optimal, but returns so many rows it still runs long.

Solution9_1: Parallelize the query so that the necessary logical and physical I/O are handled by multiple parallel threads. Watch out for excessive load spikes, and avoid object parameters that will parallelize SQL that does not need it with damaging results.

Scenario #9, Solution

```
SELECT ...
FROM
T1, T2, T3, T4, T5
WHERE T1.FKey2=T2.PKey2
AND T1.FKey3=T3.PKey3
AND T2.FKey4=T4.PKey4
AND T3.FKey5=T5.PKey5
AND T4.CCol4=:B1
AND T4.DCOL4 BETWEEN :B2 AND :B3
AND T1.CCol1=:B4
AND T3.CCol3=:B5
AND T5.CCol5=:B6
```

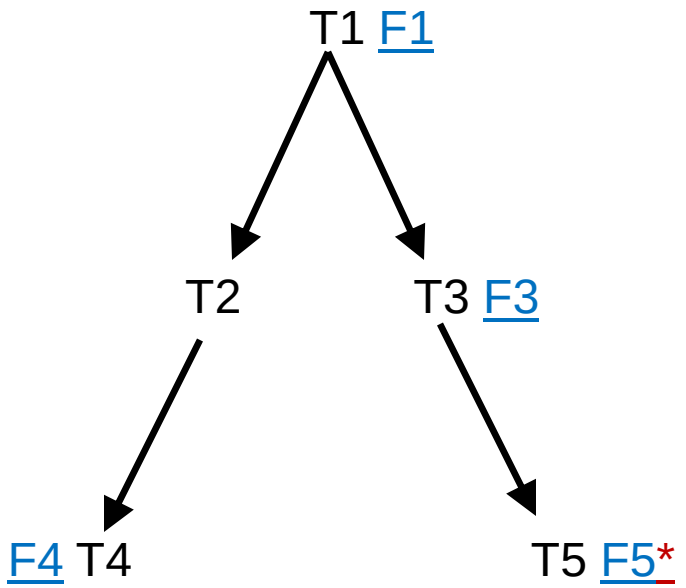


The plan is already optimal, but returns so many rows it still runs long.

Solution9_2: Consider whether the users even need to run this query, if this is an over-broad report that contains more detail than users will ever need.

Scenario #9, Solution

```
SELECT ...
FROM
T1, T2, T3, T4, T5
WHERE T1.FKey2=T2.PKey2
AND T1.FKey3=T3.PKey3
AND T2.FKey4=T4.PKey4
AND T3.FKey5=T5.PKey5
AND T4.CCol4=:B1
AND T4.DCOL4 BETWEEN :B2 AND :B3
AND T1.CCol1=:B4
AND T3.CCol3=:B5
AND T5.CCol5=:B6
```

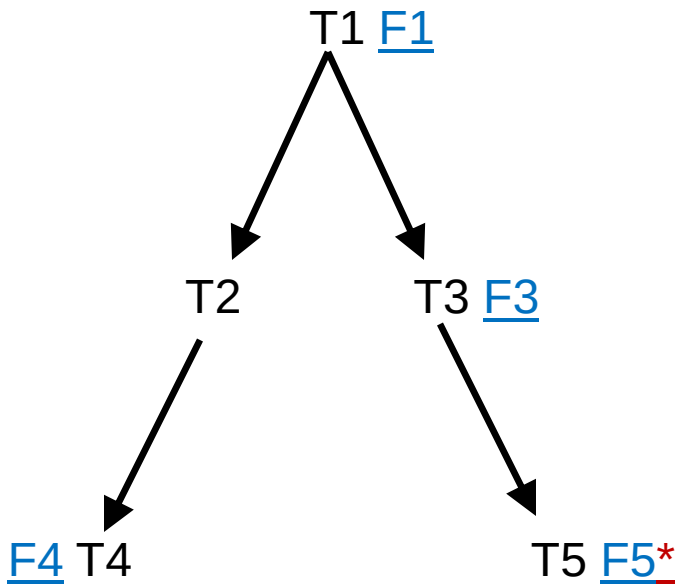


The plan is already optimal, but returns so many rows it still runs long.

Solution9_3: Consider an application-design change so that this query is no longer needed or is needed much less often if it serves some sort of middleware function.

Scenario #10

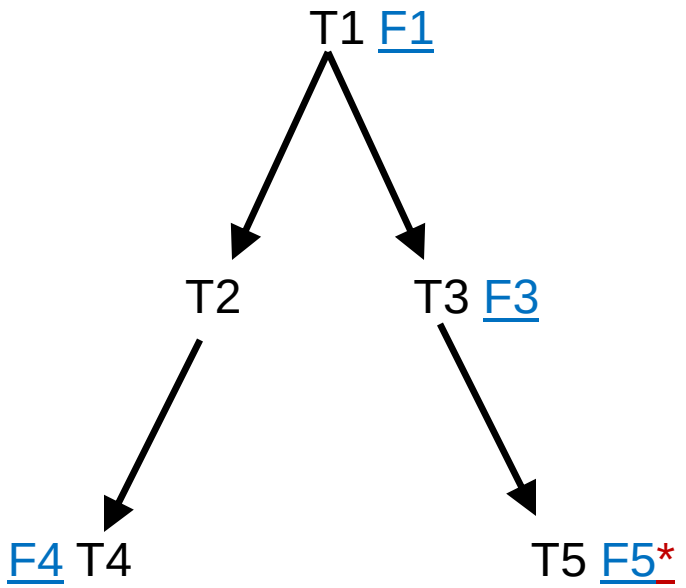
```
SELECT ...
FROM
T1, T2, T3, T4, T5
WHERE T1.FKey2=T2.PKey2
AND T1.FKey3=T3.PKey3
AND T2.FKey4=T4.PKey4
AND T3.FKey5=T5.PKey5
AND T4.CCol4=:B1
AND T4.DCOL4 BETWEEN :B2 AND :B3
AND T1.CCol1=:B4
AND T3.CCol3=:B5
AND T5.CCol5=:B6
```



The optimizer already chooses the optimally efficient execution plan. We are getting a small rowcount with a minimum number of logical and physical I/Os per row, already, and the query is already very fast, but the cumulative load and runtime are unacceptably high because this runs hundreds of thousands of times/day.

Scenario #10, Solution

```
SELECT ...
FROM
T1, T2, T3, T4, T5
WHERE T1.FKey2=T2.PKey2
AND T1.FKey3=T3.PKey3
AND T2.FKey4=T4.PKey4
AND T3.FKey5=T5.PKey5
AND T4.CCol4=:B1
AND T4.DCOL4 BETWEEN :B2 AND :B3
AND T1.CCol1=:B4
AND T3.CCol3=:B5
AND T5.CCol5=:B6
```



The plan is already optimal and fast, but this runs very frequently and is therefore still high-load.

Solution10: Eliminate the query from the application or to run it much less often. If it is running in a loop, for each row from a parent query, fold the functionality of the child query into the parent, perhaps. If it serves some sort of monitor function, consider whether the monitor could fire far less often, with longer sleeps between searches.

Observations about the Scenarios

- The CBO already delivered the best plan possible at parse time, in most scenarios.
- Where the CBO failed to deliver the best plan possible at parse time, it had good reason, given its limited information.
- Most example scenarios (8 out of 10) can be solved without overriding the CBO, ***but all scenarios still require action that the CBO cannot take without help.***

More Observations

- The 10 examples are a small sample of the possible scenarios, especially if we looked at more complex SQL.
- Problems often combine, with SQL having multiple issues at a time – solving one issue doesn't necessarily solve the whole problem.
- The solutions to several of the scenarios are expensive, so we need to be *sure* of the right answer before taking action!

The role of manual tuning in solving any scenario

- Is the optimizer choosing the best plan?
- If not, why not?
- How much worse is the current plan?
- How can we get the best plan, and at what cost?

And to answer all of the above…

- What *is* the best plan, and if the CBO isn't finding it, is there any better way to find it than manual tuning?!?

Questions?